TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

SAMI MÄKINEN

KÄYTTÖLIITTYMÄN TESTAUS ROBOT FRAMEWORKILLA

Kandidaatintyö

Tarkastaja:
projektitutkija Mikko Nurminen
7.12.2017

TIIVISTELMÄ

SAMI MÄKINEN: Käyttöliittymän testaus Robot Frameworkilla
Tampereen teknillinen yliopisto
Kandidaatintyö, 22 sivua, 1 liitesivu
Joulukuu 2017
Tietotekniikan kandidaatin tutkinto
Pääaine: Ohjelmistotuotanto
Tarkastaja: projektitutkija Mikko Nurminen

Ohjelmistojen testauksella tarkoitetaan yleisesti järjestelmällistä toimintaa, jolla pyritään löytämään ohjelmistosta virhe tai osoittamaan sen laadukkuus. Nykyään ohjelmistot ovat kuitenkin niin laajoja, että aivan kaikkea ei ole mahdollista edes testata, joten testausta suunnitellessa ja tehdessä on tärkeää miettiä, mitä kannattaa testata ja miten.

Tämän kandidaatintyön pääpaino on automaattitestauksessa ja tarkoitus onkin automatisoida web-käyttöliittymän kautta tapahtuvaa testausta Robot Framework -testiautomaatiotyökalulla. Työssä esitetään testauksen teoria testaustavoista testauksen eri vaiheisiin ohjelmistoprojekteissa.

Seuraavaksi Robot Framework esitellään ja käydään läpi sen keskeiset toiminnot yksinkertaisten esimerkkien avulla. Lopuksi sovelletaan teoriaa käytäntöön ja tuotetaan käyttöliittymän kahdelle tärkeimmälle käyttötapaukselle automaattitestit, sekä analysoidaan syntyneiden automaattitestien soveltuvuutta ja hyötyjä.

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

HTML = *Hypertext Markup Language*

TSV = *Tab-Separated Values*

reST = *reStructuredText*

API = *Application Programmable Interface*

CSV = *Comma Separated Values*

JSON = *Javascript Object Notation*

1. JOHDANTO

Tietokoneohjelmistojen toimivuus on noussut yhä tärkeämpään rooliin, sillä käyttäjämäärän kasvun takia ohjelmistossa havaittu virhe vaikuttaa usein isoon käyttäjäkuntaan. Tämän lisäksi ohjelmistot käsittelevät usein kriittistä dataa kuten henkilötietoja ja tuotantoympäristössä sattuvalla virheellä voi olla korvaamattomia seurauksia. Ohjelmistojen testaus on ohjelmistokehityksen osa-alue, joka pyrkii minimoimaan ongelmatilanteet löytämällä ohjelmistosta virheet riittävän ajoissa.

Kokonaisen järjestelmän testaus on aikaa vievä ja kallis prosessi. Manuaalinen testaus pärjää tiettyyn pisteeseen asti, jos resursseja on riittävästi, mutta järjestelmien kehittyessä alkuperäisestä versiostaan, testattavien asioiden määrä kasvaa lopulta liian suureksi. Erityisesti käyttöliittymän kautta tapahtuva manuaalinen testaus on ennen kaikkea hidasta, mutta myös hyvin virhealtista – ihminen käyttäjänä tekee virheitä ja ei aina pysty toistamaan täydellisesti aiemmin tekemiään asioita. Tässä tapauksessa käyttöliittymätestauksen osittainen automatisointi on ratkaisu, jolla edellä mainittuihin ongelmiin voidaan löytää ratkaisu.

Tämän työn tarkoitus on perehtyä käyttöliittymätestauksen automatisointiin Robot Framework -testiautomaatiotyökalulla. Luvussa kaksi tutustutaan ohjelmistotestaukseen yleisesti ja käydään läpi testaustavat ja testauksen eri tasot ohjelmistoprojektissa. Luvussa kolme esitellään testausautomaatioon työkalu Robot Framework ja tutustutaan sen eri ominaisuuksiin testien automatisoinnissa. Neljännessä luvussa sovelletaan teoriaa myyntikäyttöliittymän testauksen automatisointiin muutamassa testitapauksessa.

2. TESTAUS

Testauksella yleisesti tarkoitetaan jonkin asian kokeilemista, mutta ohjelmistojen tapauksessa kyse ei ole pelkästään kokeilemisesta, vaan järjestelmällisestä prosessista. J. Myersin [8] mukaan ohjelmistojen testaus on yritys saada ohjelma toimimaan virheellisesti, jolloin virhe korjaamalla ohjelmalle voidaan tuottaa lisää arvoa. [8, s. 1-2]

W. Hetzel [5] esittää kirjassaan ohjelmistojen testaukselle laadullisen näkökulman – testauksella pyritään osoittamaan ohjelmiston laadukkuus. Laadun yksiselitteinen määrittely on kuitenkin vaikeaa ja ohjelmistojen tapauksessa voidaan laatu jakaa kolmeen kategoriaan. Ulkoisella laadulla tarkoitetaan käyttäjälle näkyviä ominaisuuksia, kuten ohjelman toimivuutta, käytettävyyttä ja luotettavuutta. Sisäinen laatu mittaa puolestaan ohjelman tehokkuutta, testattavuutta ja rakennetta. Loput ominaisuuksista liittyvät ohjelman elinkaareen, esimerkiksi ylläpidettävyyteen ja eri osien uudelleenkäytettävyyteen. [5]

Ohjelmistojen täydellinen testaus on mahdotonta, sillä yksinkertaisestakin ohjelmasta on mahdollista löytää satojatuhansia erilaisia kombinaatioita syötteistä, joilla ohjelmaa voidaan käyttää. Testitapauksista tulisikin valita sellaiset, joilla on suurin todennäköisyys löytää virhe ohjelmasta, sillä voidaan olettaa, että virheitä aina löytyy. [8, s. 43]

2.1 Testaustavat

Ohjelman testaus voidaan pääpiirteissään jakaa kahteen kategoriaan testaustavan suhteen, lasilaatikkotestaukseen ja mustalaatikkotestaukseen. Testitavan valinta riippuu usein testattavasta ohjelmasta ja siitä, mitä halutaan testata. [8, s. 9]

2.1.1 Lasilaatikkotestaus

Lasilaatikkotestauksella tarkoitetaan testitapaa, jossa testaajalla on pääsy testattavan kohteen ohjelmakoodiin ja tietorakenteisiin. Tässä testaustavassa ohjelmakoodin perusteella voidaan rakentaa graafi, jossa kuvataan kaikki ohjelman vaiheiden keskinäiset riippuvuudet. Testitapauksia voidaan tämän jälkeen valita esimerkiksi kattamaan kaikki mahdolliset valintapolut ohjelman läpi. [13]

Ohjelman mallinnus graafina antaa mahdollisuuden matemaattisen graafiteorian ja -ohjelmien hyödyntämiseen. Tämän lisäksi lasilaatikkotestauksessa testit kirjoitetaan ohjelmakoodin perusteella, joten yhdenkään ohjelmakoodin osan ei pitäisi jäädä huomiotta. Lasilaatikkotestauksen ongelma on kuitenkin usein se, että testaus päästään aloittamaan vasta sen jälkeen, kun itse ohjelmakoodia on toteutettu. Tämä toisinaan saattaa johtaa siihen, että itse testaus jää vähemmälle huomiolle. [13]

Yksi ratkaisu edellä mainittuun ongelmaan on testivetoinen kehitys (engl. *Test Driven Development)*, jossa kehitysprosessi aloitetaan kirjoittamalla testitapaukset, joita ohjelma ei alussa läpäise. Ohjelman toteutuksen edetessä testitapauksia aletaan läpäistä ja näin saadaan samalla kuva siitä, miten ohjelman toteutus etenee. Tämän lisäksi testitapauksista saadaan usein apua itse ohjelman rakenteen suunnitteluun. Testivetoisen kehityksen soveltaminen käytännön ohjelmiin voi joskus kuitenkin olla hankalaa, sillä jotkin ohjelman ominaisuudet voivat olla hyvinkin monimutkaisia ja täten hankalia hahmottaa. [1, 4]

2.1.2 Mustalaatikkotestaus

Mustalaatikkotestauksessa testaajalla ei ole minkäänlaista tietoa testattavan kohteen sisäisestä rakenteesta. Mustalaatikkotestauksessa testitapaukset rakennetaan pelkästään määritysten perusteella; tietyllä syötteellä tulisi saada tietty lopputulos. [8, s. 9]

Mustalaatikkotestauksessa kattavan testauksen saavuttamiseksi tulisi testata kaikki mahdolliset syötteet. Tämä ei kuitenkaan ole mahdollista, joten testejä voidaan jaotella ryhmiin eri menetelmillä. [7]

Ekvivalenssiosituksessa mahdolliset syötteet jaetaan luokkiin, joiden voidaan ajatella toimivan samoin tavoin. Jos yksi luokan syötearvoista tuottaa oikean lopputuloksen, voidaan muidenkin luokan arvojen olettaa tuottavan oikean lopputuloksen. Yksinkertainen esimerkki on ohjelma, joka ottaa syötteenään kokonaisluvun ja tutkii, onko se positiivinen vai negatiivinen. Tällöin ekvivalenssiluokkia syntyy neljä: positiiviset luvut, negatiiviset luvut, nolla ja syötteet, jotka eivät ole kokonaislukuja. [13]

Raja-arvoanalyysissä valitaan syötteistä sellaiset, jotka edustavat syötteen ääriarvoja ja arvoja, jotka ovat hyvin lähellä näitä. On nimittäin havaittu, että suuri osa virheistä ilmaantuu yleensä nimenomaan raja-arvojen kohdalla. Yleinen tällainen virhe on esimerkiksi ”pienempi kuin”-vertailuoperaattorin käyttö ”pienempi tai yhtä suuri kuin”-operaattorin sijaan. [13]

Mustalaatikkotestauksen yksi suuri etu on se, että testaajalta ei vaadita kovinkaan syvällistä tuntemusta järjestelmästä eikä lainkaan tuntemusta ohjelmoinnista. Tämän lisäksi testien kirjoittaminen voidaan suorittaa ennen itse ohjelman toteuttamista. Testitapausten tekeminen voi olla kuitenkin hyvin haastavaa, mikäli määrittelydokumentit ovat puutteellisia tai epäkelvollisia. Tämä voi johtaa jopa siihen, että joissain tapauksissa mustalaatikkotestausta ei ole edes mahdollista toteuttaa [13]

2.2 Testauksen tasot

Ohjelmistojen testaukseen on useita strategioita. Kaksi ääripäätä ovat, että ohjelmisto testataan vasta kun koko ohjelma on valmis, kun taas toisessa testejä tehdään sitä mukaa kun

ohjelman osia valmistuu. Kumpikin ääripää tuottaa usein huonon lopputuloksen, ja ohjelmistoprojektista riippuen testaus onkin jaettu tasoihin. [9, s. 473] Tasoja on neljä: yksikkötestaus (engl. *unit testing*), integrointitestaus (engl. *integration testing*), järjestelmätestaus (engl. *system testing*) ja hyväksymistestaus (engl. *acceptance testing*). Kuvassa 1 on esitetty testauksen eri tasot.

Kuva 1: Testauksen tasot

Seuraavissa neljässä aliluvussa tutustutaan testauksen eri tasoihin ja otetaan samalla kantaa, kumpaa testaustapaa kullakin tasolla yleensä käytetään. Luvussa 2.3 yhdistetään eri tasot ohjelmistokehityksen prosessin eri vaiheisiin.

2.2.1 Yksikkötestaus

Yksikkötestaus on neljästä tasosta kaikkein alin ja keskittyy pienimpään kokonaisuuteen. Testattavat yksiköt voivat olla ohjelman yksittäisiä komponentteja tai moduuleja eli luokkia. Yksikkötestauksessa keskitytään ainoastaan kunkin komponentin sisäisiin toimintoihin, kuten tietorakenteisiin ja komponentin toimintalogiikkaan. Komponentteja voidaan tällöin testata samanaikaisesti, niiden ollessa vielä toisistaan riippumattomia tässä vaiheessa. Yksikkötestaus on aina lasilaatikkotestausta, sillä testauksen kohteena on nimenomaan tietorakenteet ja yksikön sisäinen toiminnallisuus. [9, s. 473]

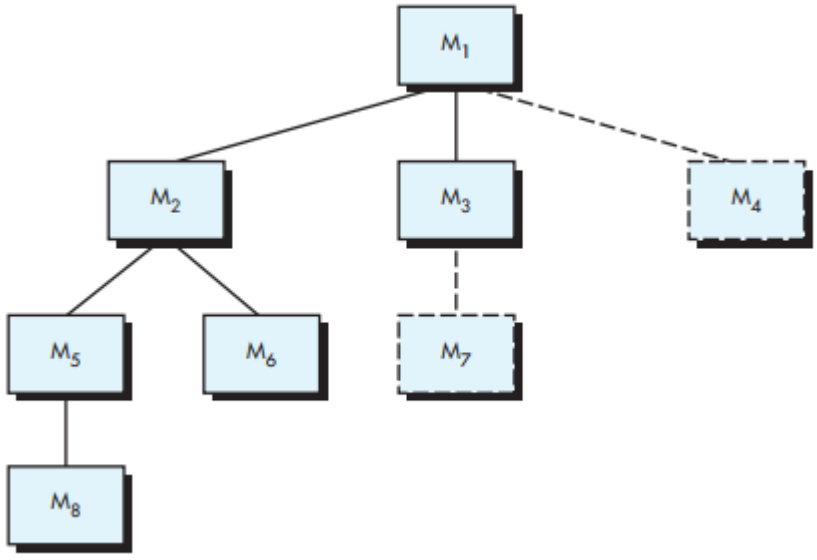
2.2.2 Integrointitestaus

Integrointitestaus on yksikkötestauksen jälkeen seuraava taso lähempänä valmiin ohjelman testausta. Integrointitestauksen tarkoitus on todeta yksinään toimivien ohjelman komponenttien toimivuus yhdessä. Kaikkia komponentteja ei tule suoraan yhdistää keskenään, sillä kokonaisuudesta tulee usein liian monimutkainen, vaan integrointi on syytä

tehdä vaiheittain. Komponenttien integroinnin vaiheistamiseen on kaksi vaihtoehtoa; ylhäältä alas -integrointi (engl. *Top-Down Integration*) ja alhaalta ylös -integrointi (engl. *Bottom-Up Integration*). Jokaisen integroimisvaiheen jälkeen suoritetaan varsinainen testaus. Jos testauksessa löytyy virhe, niin se korjataan ja voidaan jatkaa integrointia seuraavalle tasolle. [9, s. 475-477]

Ylhäältä alas- integroinnissa ohjelman komponentit yhdistetään toisiinsa niin, että aloitetaan ohjelman hierarkiassa ylimmältä tasolta eli pääohjelmasta. Pääohjelmaan voidaan muut ohjelman osat yhdistää etenemällä hierarkiaa syvyyteen ensin -integroinnilla (engl. *depth-first integration*), jolloin edetään kutakin polkua mahdollisimman pitkälle ennen kuin seuraavia polkuja lisätään. Kuvan 2 tapauksessa osat yhdistettäisiin pääohjelmaan (M1) järjestyksessä M2, M5, M8, M6, M3, M7 ja M4. Toinen vaihtoehto on edetä leveyteen ensin -integroinnilla (engl. *breadth-first integration*), jolloin lisätään ensin kaikki suoraan pääohjelman alla olevat moduulit ja sen jälkeen niiden alla olevat ja niin edelleen. Tällöin komponentit lisättäisiin pääohjelmaan järjestyksessä M2, M3, M4, M5, M6, M7 ja M8. [9, s. 476]

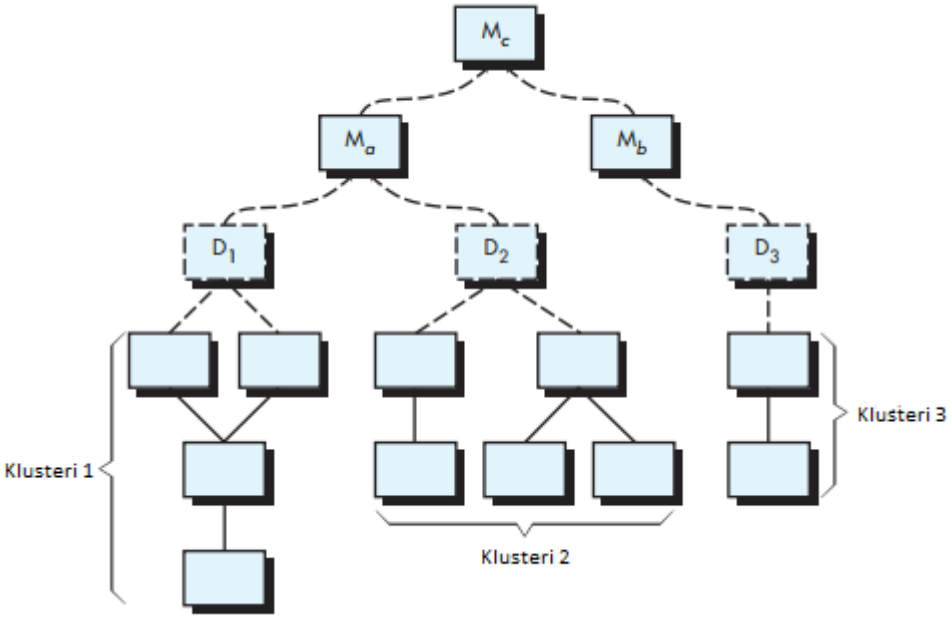
Ylhäältä alas- integroinnissa puuttuva komponentti tarvitsee korvata tyngällä (engl. *stub*), joka on hyvin yksinkertaistettu versio varsinaisesta ohjelman osasta. Tynkä korvaa oikean komponentin, kunnes itse komponentti lisätään rakenteeseen. [9, s. 475]



Kuva 2: Ylhäältä alas -integrointi [9, s. 477]

Alhaalta ylös -integroinnissa lähdetään liikkeelle alimman tason komponenteista ja yhdistetään niitä keskenään klustereiksi (engl. *cluster*), jotka toteuttavat jonkin pienen osan ohjelman toiminnallisuudesta. Alhaalta ylös -integroinnissa klusterit ja yksittäiset komponentit tarvitsevat erillisiä ajureita (engl. *driver*), kun erillistä suorittavaa pääohjel-

maa ei ole. Kuvassa 3 on esitetty alhaalta ylös -integrointi. Ensin luodaan alimman tason komponenteista kolme klusteria, ja niille suorittavat ajurit D1, D2 ja D3. Kunkin klusterin testauksen jälkeen klusterit voidaan yhdistää ylemmän tason komponentteihin ja niin edelleen. Lopulta päädytään kokonaiseen ohjelmaan. [9, s. 478]



Kuva 3: Alhaalta ylös -integrointi [9, s. 478] suomennos

Ylhäältä alas -integroinnin suurin etu on se, kun lähdetään liikkeelle pääohjelmasta, niin koko ohjelma on koko ajan käytettävissä, kun taas alhaalta ylös -integroinnissa koko kuva ohjelmasta saadaan vasta kun kaikki osat on yhdistetty. Ylhäältä alas -integrointi vaatii kuitenkin aina tynkien käyttöä, mikä on yksi ylimääräinen toteutettava osa lisää joka komponentin kohdalla. [6, s. 142]

2.2.3 Järjestelmätestaus

Järjestelmätestaus on seuraava vaihe integrointitestauksen jälkeen ja se onkin usein kaikkein pisin kestoltaan. Järjestelmätestauksessa testattava kohde on koko ohjelmisto, mutta myös siihen vaikuttavat ulkoiset tekijät, kuten esimerkiksi laitteisto ja ihmiset tulee mahdollisuuksien mukaan ottaa huomioon. [9, s. 486] Järjestelmätestaus aloitetaan usein testaamalla järjestelmä määritysten perusteella mustana laatikkona – varmistetaan että ohjelmassa on tarvittavat ominaisuudet ja ne toimivat niin kuin pitääkin. [5]

Järjestelmätestauksessa testataan usein myös määritysten lisäksi muilla tavoilla, kuten esimerkiksi mittaamalla järjestelmän suorituskykyä, palautumista virhetilanteista ja turvallisuutta. Suorituskykyä testatessa järjestelmään voidaan simuloida esimerkiksi useita yhtäaikaisia käyttäjiä ja todeta järjestelmän selviytyminen ”ruuhkasta”. Ohjelmistojen tu-

lee myös osata palautua mahdollisista poikkeustilanteista, esimerkiksi tietoliikennekatkoksesta. Palautumista voidaan testata yksinkertaisesti simuloimalla virhetilanne. [9, s. 486-487]]

Tietoturvan testaus on tärkeä osa-alue varsinkin järjestelmissä, jotka käsittelevät kriittistä dataa, kuten esimerkiksi henkilötietoja. Tietoturvaa voidaan testata esimerkiksi simuloimalla hyökkäys palveluun. [9, s. 487] Tietoturvan ja turvallisuuden testaukseen ei perehdytä tässä työssä syvemmin aiheen laajuuden vuoksi.

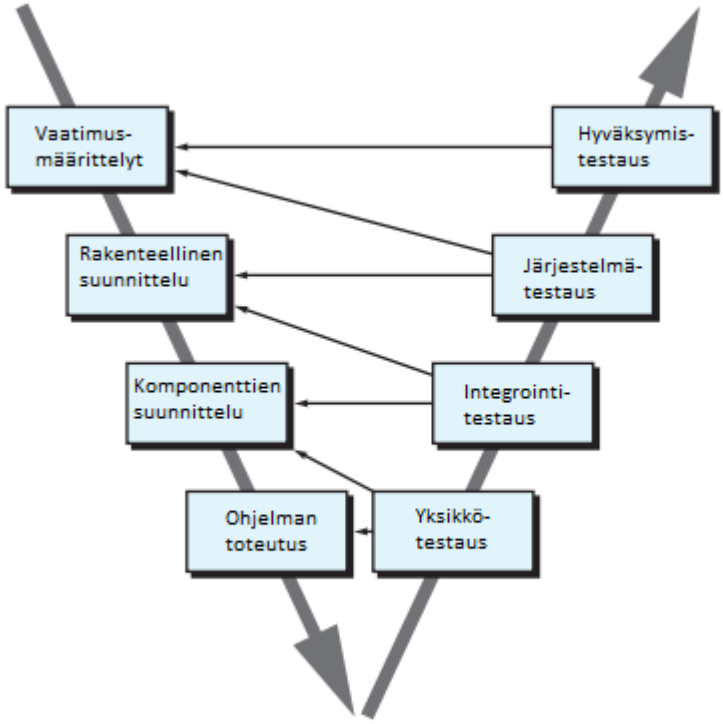
2.2.4 Hyväksymistestaus

Hyväksymistestaus on ohjelmistotestauksen viimeinen vaihe, ja sen suorittaa usein ohjelmiston loppukäyttäjä. Hyväksymistestauksen testitapaukset saadaan yleensä käyttäjätarinoista (engl. *User Story*), joissa kuvataan jonkin toiminnallisuuden käyttö alusta loppuun. [9, s. 75] Testitapausten suorituksessa voidaan usein hyödyntää vanhaa järjestelmää, jos uusi testattava järjestelmä on korvaamassa vanhaa. Voidaan esimerkiksi syöttää kumpaankin järjestelmään samat syötteet ja verrata toimivatko uusi ja vanha järjestelmä samalla tavalla. [5]

Hyväksymistestaus voidaan jakaa alfa- ja betatestaukseen. Alfatestauksessa ohjelmiston kehittäjä on testauksessa mukana ja testit suoritetaan erillisessä kontrolloidussa testiympäristössä. Betatestauksessa testaukseen käytetään suoraan joko tuotantoympäristöä, tai sitä vastaavaa ympäristöä. Löydetyt viat ilmoitetaan ohjelmiston kehittäjälle ja korjausten jälkeen ohjelmisto on käytännössä valmis julkaistavaksi. [9, s. 485]

2.3 Testaus ohjelmistoprojektissa

Luvuissa 2.2.1-2.2.4 esitetyt neljä testauksen tasoa voidaan yhdistää ohjelmistoprojektin vaiheisiin niin sanotun V-mallin avulla. Kuvassa 4 ohjelmistoprojekti alkaa mallin vasemmalta reunalta ylhäältä vaatimusmäärittelyistä. Ohjelman kehitys siirtyy vasenta reunaa alaspäin. Itse ohjelman toteutuksen alettua, testaus voidaan aloittaa yksikkötestauksesta ja edetä vaihe kerrallaan kohti lopullista hyväksymistestausta. [6, s. 81-83]



Kuva 4: V-malli [9, s. 42] suomennos

V-mallin vasenta reunaa edetessä jokaisessa vaiheessa suunnitellaan jokaista vaihetta vastaavat testit, jotka suoritetaan myöhemmin. Testauksen suunnittelua ja toteutusta käsitellään omana lukunaan luvussa 2.4.

2.4 Testauksen suunnittelu ja toteutus

Testauksen huolellinen suunnittelu on osa testauksen järjestelmällisyyttä. Tavoitteena on testata oleelliset asiat riittävällä tasolla, mutta kuitenkaan resursseja tuhlaamatta. Testisuunnitelmalla tarkoitetaan etukäteen tehtyä suunnitelmaa, joka sisältää muun muassa tiedot siitä, mitä on tarkoitus testata, miten testataan, ja millä aikataululla. [8, s. 146]

Vastattaessa kysymykseen ”Mitä on tarkoitus testata?” on usein tarpeen priorisoida testattavia asioita – kaikkea ei kuitenkaan ehditä testaamaan. Riskianalyysissä arvioidaan ominaisuuksien tärkeys niiden käyttötaajuuden ja kriittisyyden perusteella. Riskianalyysi voidaan toteuttaa taulukkona, johon on listattu toiminto ja ominaisuus ja siihen liittyvä käytön taajuus ja kriittisyys. [6, s. 241-245]

Kuvassa 5 on listattuna kolme puhelimeen liittyvää ominaisuutta ja kunkin kohdalla on arvioitu käytön taajuutta asteikolla 1-5 (1=hyvin vähän käytetty, 5=erittäin paljon käytetty) ja onnistumisen kriittisyyttä asteikolla 1-5 (1=ei kriittinen, 5=erittäin kriittinen). Näiden kahden arvon tulo kertoo ominaisuuden prioriteetin, jonka mukaan ominaisuudet on helppo järjestää tärkeimmästä vähimmän tärkeimpään.

Ominaisuus	Käytön taajuus (1-5)	Onnistumisen kriittisyys (1-5)	Prioriteetti (tulo)
Puhelun soitto	5	3	15
Hätäpuhelu	1	5	5 (kuitenkin kriittinen…)
Soittajan nimen näyttö	5	1	5

Kuva 5: Riskianalyysi [6, s. 244]

Itse testausvaihe suoritetaan usein testilokin avulla, johon kirjataan ylös suoritetut testit, testien tekijät ja tulokset. Testilokin tarkoitus on helpottaa testauksen seurantaa ja hallinnointia. Testauksen hallintaan on myös erillisiä testauksenhallintajärjestelmiä, joihin on usein liitetty myös niin kutsuttu vikatietokanta. Vikatietokantaan kirjataan testauksen aikana löydetyt viat ja korjausten tilanne, mikä helpottaa vikatilanteen seurantaa [6, s. 273-274]]

2.5 Automaattitestaus

Automaattitestauksella tarkoitetaan testausta, jonka suorittamiseen tarvitaan vain vähän tai ei ollenkaan manuaalista työtä. Testiautomaatiolla voidaan M. Fewsterin [3] mukaan parhaassa tapauksessa säästää jopa 80% testaukseen kuluvissa kuluissa, mutta lähes aina testauksen automaatio johtaa ohjelmiston laadun paranemiseen. Automatisoidut testit voidaan usein suorittaa murto-osassa ajasta manuaalisiin testeihin verrattuna, joten testaukseen kuluva aika pienenee huomattavasti. [3, s. 4]

Kaikkea testausta ei kuitenkaan edes kannata automatisoida, sillä automaation hyödyt saadaan jo usein automatisoimalla pieni osa testeistä. M. Fewsterin [3, s. 230-231] mukaan seuraavat neljä asiaa kannattaa huomioida valittaessa, mitä kannattaa automatisoida:

1. Testit, jotka testaavat kaikkein tärkeimpiä ominaisuuksia
2. Testit, jotka on helpoin automatisoida
3. Testit, joilla suurimmalla todennäköisyydellä löydetään virhe
4. Testit, jotka suoritetaan kaikkein useimmin.

2.5.1 Automaatiotestauksen työkalut

Testauksen automatisointiin on useita eri työkaluja ja tarkoitukseen sopivan työkalun valinta onkin tärkeää. Työkalua valittaessa tulisi aina suorittaa huolellinen taustatyö, jossa voidaan esimerkiksi yrittää arvioida nykyisen manuaalitestauksen hintaa, löytämättä jääneiden virheiden kustannuksia ja uuden automaatiotyökalun hankinta- ja ylläpitohintaa. Näiden asioiden perusteella voidaan tehdä kannattavuuslaskelma. [3, s. 258-259]]

Automatisoidussa testissä suoritetaan yleensä skripti (engl. *script*) ja skriptin luontiperiaate jaottelee testiautomaatiotyökalut ryhmiin. Lineaarinen skripti (engl. *linear script*) on kaikkein yksinkertaisin ja siinä nauhoitetaan kaikki testissä suoritettavat toiminnot, kuten esimerkiksi näppäimenpainikkeet ja hiiren liikkeet. Lineaaristen skriptien suurin ongelma on niiden ylläpidettävyys – joka kerta, kun jokin käyttöliittymässä muuttuu, skriptit on nauhoitettava uudestaan. [3, s. 75-77]

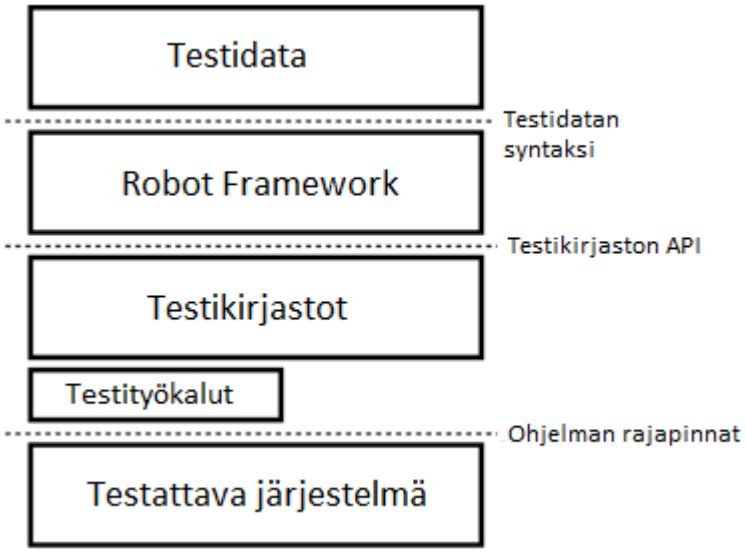
Rakenteellinen skripti (engl. *structural script*) on jaettu osiin, joita voidaan kutsua erikseen. Tämä mahdollistaa silmukka- ja valintarakenteiden testaamisen ja skriptissä voidaan jopa tutkia yksinkertaisia ehtoja ja toimia niiden mukaan. Rakenteellisen skriptin ongelma onkin usein, että testitapausten testidata on yhä ”kovakoodattu” skriptiin. Rakenteellista skriptiä hyvin lähellä on jaettu skripti (engl. *shared script*), jossa samaa skriptiä voidaan kutsua useasta eri ohjelmasta. Tämä vähentää usein tarvittavien skriptien määrää. [3, s. 78-79]

Datalähtöisessä skriptissä (engl. *data-driven script*) kunkin testin data säilytetään erillisessä tiedostossa, jolloin testidatan muokkaaminen on helpompaa. Vielä älykkäämpi malli on avainsanalähtöinen skripti (engl. *keyword-driven script*), jossa testattavan järjestelmän toiminnot on jaettu niin sanotuiksi avainsanoiksi. Kullekin avainsanalle luodaan toteutus ja tämän jälkeen niitä on helppo kutsua testin osana. Tämä helpottaa itse testiskriptin luomista, sillä avainsanalähtöiset skriptit ovat usein luettavissa lähes selkokielisinä. [3, s. 83, 89-91]

Tässä työssä käytettävä Robot Framework kuuluu avainsanalähtöisiin testaustyökaluihin. Robot Framework on jo käytössä kohdeyrityksessä ja sen on todettu soveltuvan hyvin testauksen tarpeisiin. Tämän lisäksi valinnan pohjalla on halu oppia käyttämään työkalua.

3. ROBOT FRAMEWORK

Robot Framework on Python-ohjelmointikieleen perustuva avoimen lähdekoodin testausautomaatiotyökalu. Työkalun on kehittänyt Pekka Klärck yhdessä Nokia Networksin kanssa vuonna 2005. Vuonna 2008 julkaistiin nykyinen avoimen lähdekoodin versio. Avainsanapohjaisuus ja testikirjastojen käyttö tekevät työkalusta hyvin monipuolisen ja laitteistosta sekä ohjelmistosta riippumattoman. [2, 11] Robot Frameworkin arkkitehtuuri on esitetty kuvassa 6.

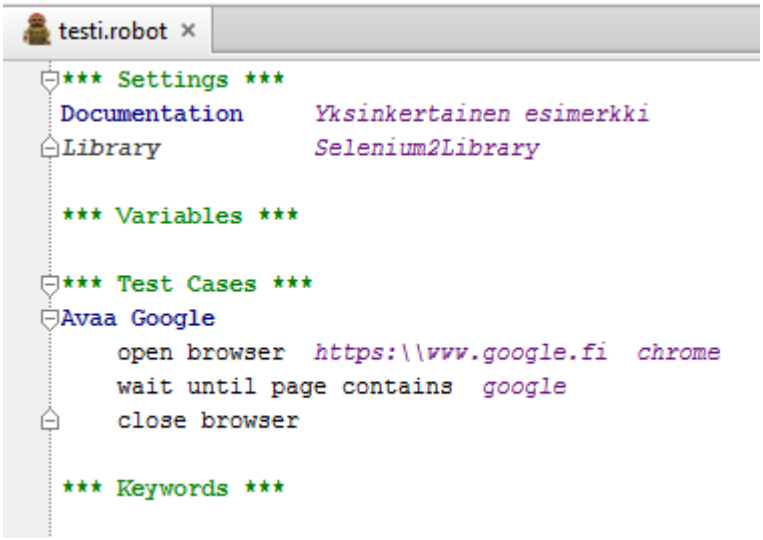


Kuva 6: Robot Frameworkin arkkitehtuuri [11] suomennos

Testidata pitää sisällään testitapauksessa käytettävän datan, esimerkiksi syötteet, syötetiedostot, ohjelmapolut ja testitapauksen toiminnot. Robot Framework itsessään suorittaa testidatan mukaisen testitapauksen testikirjastojen API:n kautta. Testikirjastot voivat kutsua suoraan testattavan järjestelmän rajapintoja tai käyttää erilaisia työkaluja. [11]

3.1 Testidata ja sen muokkaus

Testidatan syntaksille on neljä vaihtoehtoa: HTML, TSV, reST tai tekstimuoto. Yksinkertaisin neljästä syntaksista on puhdas tekstimuoto, jossa tekstin erottimena käytetään kahta välilyöntiä. Tekstimuotoista testidataa voidaan muokata millä tahansa tekstieditorilla. PyCharm on Pythonille luotu kehitysympäristö, johon saa myös oman laajennuksen Robot Frameworkille. Kuvassa 7 on esitetty yksinkertainen testidatatiedosto, joka on tekstimuodossa.



Kuva 7: Yksinkertainen testidatatiedosto

Testidata jakaantuu neljään osaan: asetuksiin, muuttujiin, testitapauksiin ja avainsanoihin. Asetuksissa (kuvassa 7 *** *Settings* ***) määritellään testin tarvitsemat kirjastot ja lähdetiedostot, sekä voidaan mahdollisesti kertoa testin sisällöstä *Documentation* -toiminnolla. Testissä muuttujat löytyvät omasta osiostaan (kuvassa 7 *** *Variables* ***), mutta muuttujia voidaan myös ladata erillisestä tiedostosta esim. CSV- tai JSON -muodossa. [11]

Itse testitapauksen askeleet löytyvät osiosta *** *Test Cases* ***, johon voidaan sisällyttää useampiakin testitapauksia. Testitapauksen alle syötetään sisennettynä vaiheet. Kuvan 7 esimerkissä kutsutaan avainsanoja *open browser, wait until page contains* ja *close browser*. Omia avainsanoja voi määritellä *** *Keywords* *** -osiossa ja omien avainsanojen toteutuksen voi tehdä itse Python- tai Java- kielillä. [11]

3.2 Kirjastot

Robot Frameworkin kirjastot sisältävät matalimman tason avainsanat, jotka hoitavat suurimman osan interaktioista testattavan järjestelmän kanssa. Kirjaston saa käyttöön joko käyttämällä asetusten *Library* -toimintoa tai kutsumalla valmista avainsanaa *Import Library*. Robot Frameworkissa on 10 standardikirjastoa, joista *BuiltIn* -kirjasto on ilman erillistä lisäystä aina käytössä. *BuiltIn* sisältää kaikkein yleisimmät ja eniten käytetyimmät avainsanat. [11] Muita olennaisia valmiita kirjastoja ovat muun muassa:

Collections – Kirjasto Pythonin lista ja sanakirjarakenteiden käsittelyyn

DateTime – Kirjasto päivämäärän ja kellonaikojen hallintaan

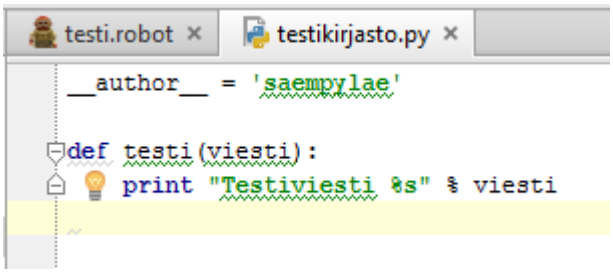
Screenshot – Kirjasto kuvaruutukaappausten ottamiseen

String – Kirjasto merkkijonotoimintojen suorittamiseen

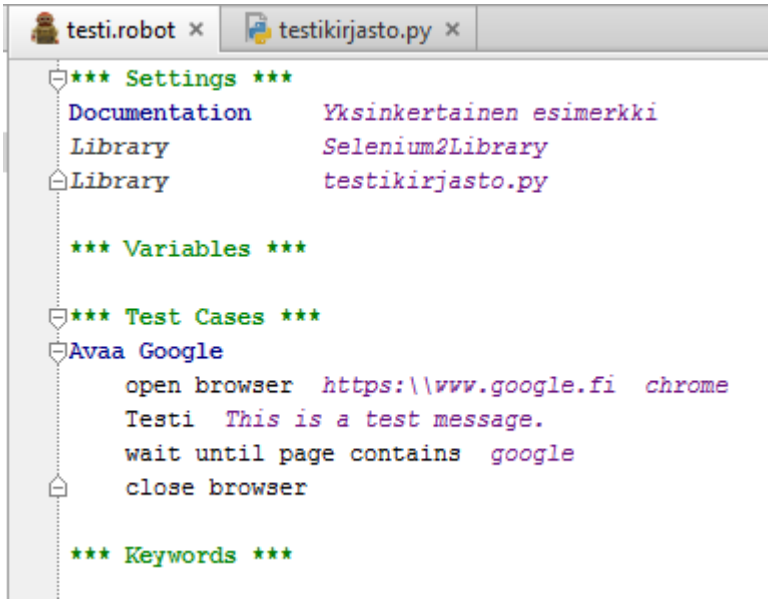
XML – Kirjasto XML-tiedostojen käsittelyyn

Standardikirjastojen lisäksi Robot Frameworkin käyttöön saa itsetehtyjä ja muiden tekemiä kirjastoja. Valmiista kirjastoista löytyy kattavat dokumentaatiot ja käyttöohjeet. [10]

Kuvassa 8 on yksinkertainen esimerkki oman kirjaston ja avainsanan luomisesta Python-kielellä ja kuvassa 9 on muokattu aiemmin käytetty testitiedosto käyttämään luotua kirjastoa



Kuva 8: Oman kirjaston luonti



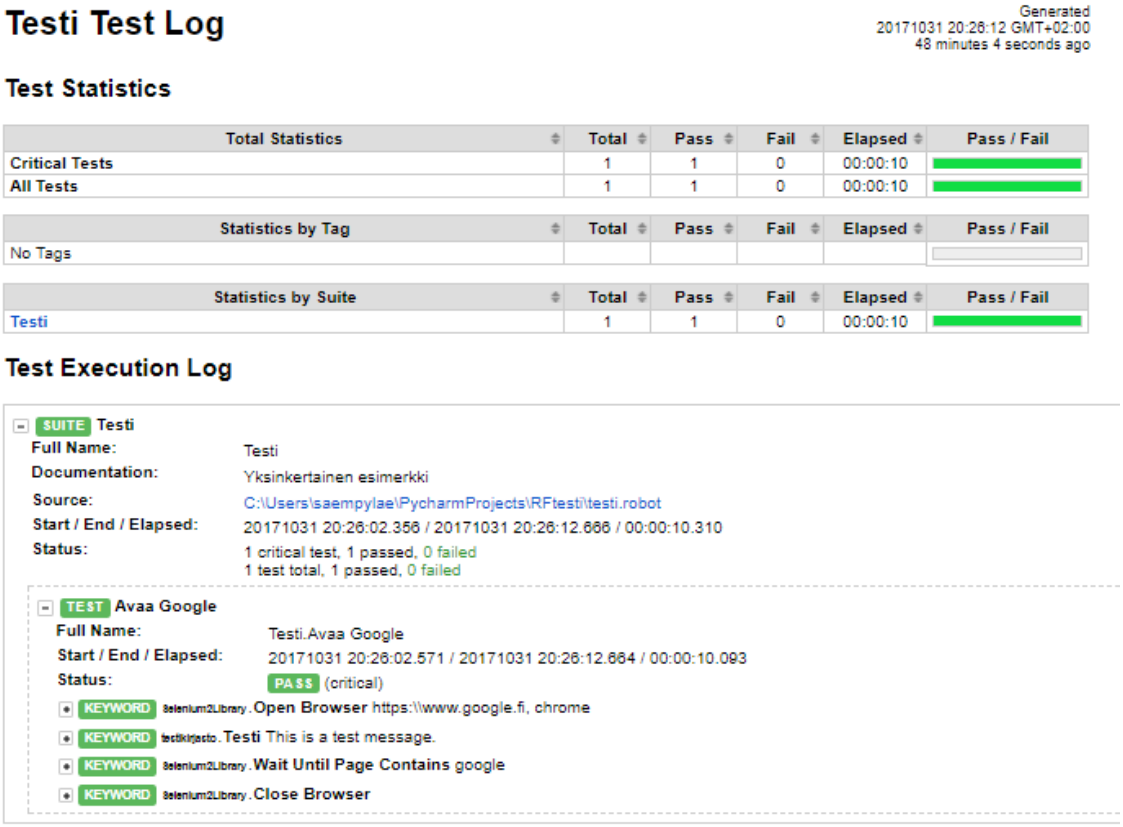
Kuva 9: Oman kirjaston käyttöönotto

Itsetehty kirjasto saadaan yksinkertaisesti käyttöön lisäämällä asetuksiin *Library*-avainsana, jolloin kaikki kirjaston sisällä olevat avainsanat tulevat automaattisesti testitiedoston sisällä käytettäviksi.

3.3 Testien tulokset

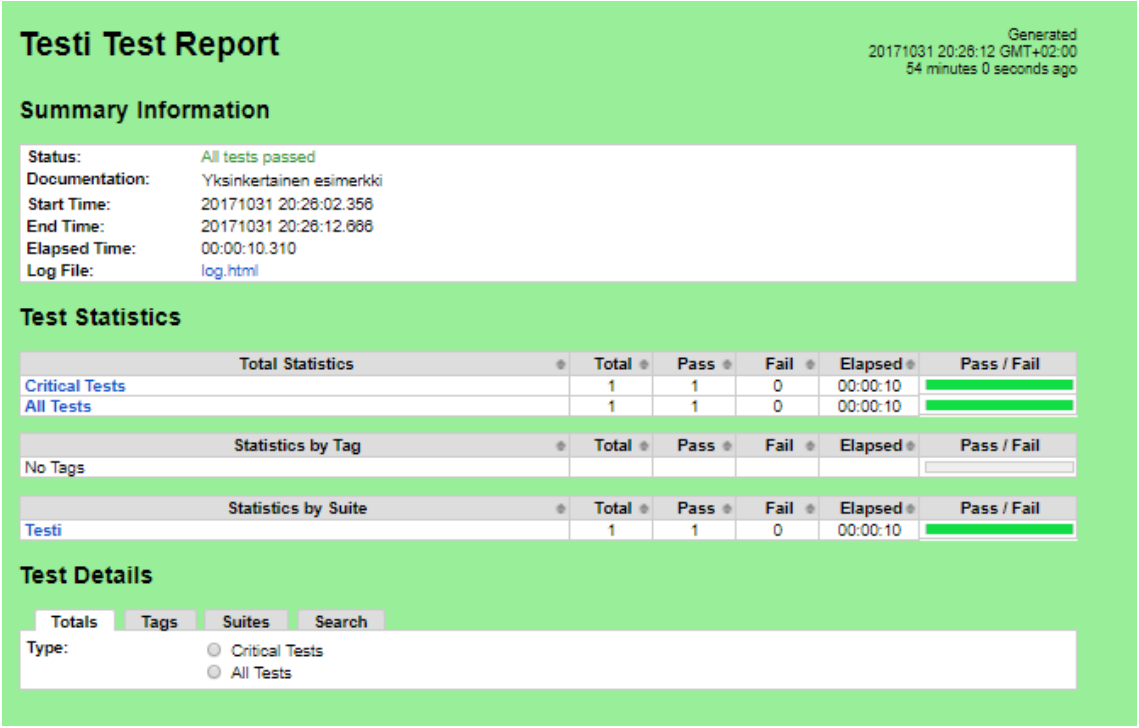
Testitapausten tuloksien esittämiseen ja erilaisten lokitiedostojen muodostumiseen on Robot Frameworkissa runsaasti toimintoja. Jokaisesta suoritetusta testistä syntyy HTML-muotoinen lokitiedosto, joka pitää sisällään yksityiskohtaiset tiedot suoritetuista testiaskeleista. Yksityiskohtaisen lokin tuottaminen on erityisen hyödyllistä, jos testin vaiheita tarvitsee tarkastella tarkemmin, esimerkiksi vikatilanteen ilmettyä. [11]

Kuvassa 10 on aiemman yksinkertaisen esimerkin tuottama lokitiedosto. Lokitiedostossa näkyy kaikki avainsanat suoritusjärjestyksessä ja kunkin kohdalla suorituksen onnistuminen. Testin vaiheisiin kuluneesta ajasta voidaan suoraan verrata koko testin tehokkuutta esimerkiksi manuaaliseen vastineeseen.



Kuva 10: Esimerkin tuottama testiloki

Vähemmän yksityiskohtia sisältävä vaihtoehto testilokille on tuotettu HTML-muotoinen *report* -tiedosto, jossa on kokonaiskuva ajetusta testistä. Raportin väritehosteet kertovat suoraan koko testin onnistumisesta. Kuvassa 11 on aiemman esimerkin tuottama raportti.



Kuva 11: Esimerkin tuottama testiraportti

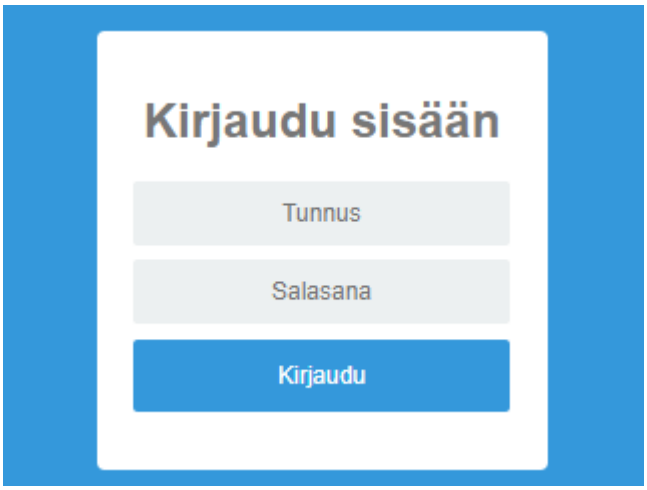
Mikäli jokin testitiedoston avainsanoista epäonnistuisi, niin kyseisen avainsanan väri muuttuisi punaiseksi lokissa ja tiedot avaamalla pääsisi tarkastelemaan kuvakaappausta tilanteesta, johon testi pysähtyi. Vastaavasti raportin taustaväri muuttuisi punaiseksi.

3.4 Käyttöliittymätestaus Selenium2 -kirjastolla

Käyttöliittymätestauksella tarkoitetaan yleisesti testattavan järjestelmän käyttöliittymän kautta tapahtuvaa testausta. Robot Frameworkin omissa kirjastoissa ei ole tarkoitukseen sopivaa kirjastoa, mutta Selenium2 -kirjasto tuo käyttöön etenkin web-sovellusten testaukseen tarvittavat ominaisuudet. Kirjasto perustuu Selenium -selainautomaatiotyökaluun, jolla voidaan itsessään nauhoittaa ja toistaa yksinkertaisia skriptejä.[2]

Selenium2 -kirjasto sisältää avainsanat kaikkein yleisimpiin toimintoihin, joita web-käyttöliittymän käyttöön tarvitaan. Toiminnallisuuksiin kuuluu muun muassa erilaisten elementtien klikkaus, tekstin syöttö, valintakenttien valinta ja painikkeiden painaminen. Web-sivun eri elementtien tunnistukseen kirjasto käyttää niin kutsuttuja paikantimia (engl. *locator*), joilla sivulta voidaan etsiä elementtiä esimerkiksi elementin tunnisteen (engl. *id*) tai nimen avulla. Elementtien paikannuksen avulla voidaan asettaa runsaasti ehtoja testin suoritukseen, esimerkiksi ”Odota kunnes elementistä *id=text* löytyy teksti xxx” tai ”Sivun tulisi sisältää kuva xxx”. [12]

Kuvissa 12 ja 13 on kirjoitettu yksinkertaiselle sisäänkirjautumiselle automatisoitu testi käyttäen Selenium2 -kirjaston avainsanoja. Testissä käytetty sisäänkirjautumissivu on toteutettu Pythonin Flask-frameworkilla, jolla on helppo luoda yksinkertaisia web-sivuja.



Kuva 12: Kuvakaappaus luodusta sisäänkirjautumissivusta

Kuva 13: Kuvakaappaus Selenium2 -esimerkin testidatasta

Esimerkin testitapaus avasi Google Chrome -selaimen ja, kun teksti ”Kirjaudu Sisään” ilmeni näytöllä, niin tunnus- ja salasanakenttiin kirjoitettiin annetut tiedot. Tämän jälkeen painettiin ”Kirjaudu” -painiketta ja tutkittiin, löytyikö sivulta tieto onnistuneesta sisäänkirjautumisesta. Koko testin suorittamiseen kului aikaa vain noin 8 sekuntia.

4. KÄYTTÖLIITTYMÄN TESTAUS ROBOT FRAMEWORKILLA

Tässä luvussa esitellään testattava järjestelmä ja sen yleisin käyttötapaus. Tämän jälkeen valitaan testitapaukset, kirjoitetaan Robot Frameworkille vastaavat testiskriptit ja lopuksi analysoidaan testien tuottamia tuloksia.

4.1 Testattava järjestelmä

Testattava järjestelmä on suomalaisen vahinkovakuutusyhtiön lemmikkivakuutusten myyntiin tarkoitettu webkäyttöliittymä. Sovellus on jo tuotantokäytössä, mutta testiversion testaus suoritetaan tuotantoympäristöä vastaavassa erillisessä testiympäristössä. Sovelluksen käyttäjät ovat yrityksen yhteistyökumppanin myyjiä ja sovellusta käytetään ainoastaan erillisessä sisäverkossa.

Webkäyttöliittymän kautta on mahdollista ostaa kissalle tai koiralle vakuutus. Yleisin käyttötapaus koostuu seuraavista vaiheista: sisäänkirjautuminen, lemmikin tietojen täyttäminen, vakuutuksen sisällön valinta, yhteystietojen haku henkilötunnuksella ja oston vahvistaminen.

4.2 Testitapausten valinta

Automatisoitaviksi testitapauksiksi valikoitui kummankin lemmikkityypin vakuutuksen onnistunut osto, sillä ne ovat ylivoimaisesti tärkeimmät ja yleisimmät websovelluksen käyttötavat. Tarkoitus on luoda kummallekin lemmikkityypille helposti käytettävät automaattitestit, joita voidaan käyttää erillisillä syötetiedostoilla. Syötetiedostoilla voidaan vaihdella vakuutuksen hintaan vaikuttavia tekijöitä (esimerkiksi rotu, lemmikin ikä tai vakuutuksen sisältö) myöhempää tarkistelua varten.

Kehitystiimin kanssa käydyssä keskustelussa tuli ilmi, että sovelluksen myöhempää kehitystä varten yksinkertaisia testejä voisi käyttää testaamaan sovelluksen yleinen toimivuus ennen muun testauksen aloittamista. Robot Framework on myös helppo asettaa testaamaan sovellus eri selaimilla.

4.3 Testien kirjoittaminen ja suoritus

Ennen testien kirjoittamisen aloittamista Robot Framework ja muut tarvittavat ohjelmat piti asentaa. Python-tulkin asennuksessa tärkeä huomio oli valita versio 2.7.X, sillä Robot

Framework ei ole vielä Python 3 -yhteensopiva. Testikoodin editoimiseen löytyi jo valmiiksi asennettuna PyCharm, joka on erityisesti Pythonille suunniteltu editori. Pythonin asennuksen mukana tuli PIP, joka on kätevä asennustyökalu Pythonin liitännäisille.

Robot Framework ja Selenium2Library voitiin helposti asentaa komennoilla:

```
pip install RobotFramework

pip install RobotFramework-Selenium2Library
```

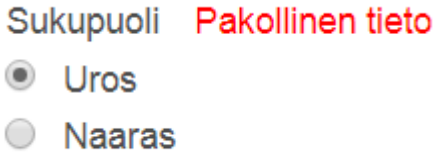
Lopuksi piti vielä asentaa halutuille selaimille ajurit, jotka löytyivät pienen etsinnän jälkeen. Kaiken kaikkiaan tarvittavien ohjelmien asennukseen kului noin 30 minuuttia, jonka jälkeen Robot Framework oli täysin käyttövalmis.

4.3.1 Ensimmäinen versio testistä

Testin kirjoittaminen aloitettiin sisäänkirjautumisesta ja edettiin vaihe kerrallaan eteenpäin. Henkilötunnuksen generointiin käytettiin jo yrityksessä aiemmin luotua omaa Python-kirjastoa. Chrome-selaimen “Tarkista”-ominaisuus osoittautui hyvin käteväksi työkaluksi elementtien tunnisteiden (engl. *id*) etsimiseen testitapauksia varten. Elementtien, joilla ei ole tunnistetta (esim. React-elementit) lokatointi osoittautui hankalaksi ja käyttöön otettiin xPath()-funktio, jolla voidaan hakea elementtiä myös esimerkiksi sen sisältämän tekstin perusteella. Myöhempää kehitystä varten sovellustiimille ilmoitettiin tarve lisätä puuttuvat tunnisteet kaikille elementeille.

Ensimmäisen vaiheen testitiedosto loi koiralle vakuutuksen ja tutki lopuksi löytyykö onnistuneesta ostosta syntyvä ”Kiitos”-teksti. Tässä vaiheessa muuttujia ei vielä luettu syötetiedostosta vaan ne oli kirjoitettu suoraan testidataan.

Ensimmäisen version testin kirjoittamiseen kului aikaa noin kolme tuntia ja muutaman onnistuneen ajon jälkeen testattiin testin ajoaikaa. Viiden testin suoritusaikojen keskiarvoksi tuli noin 13,4 sekuntia. Yksi testikerroista ei mennyt loppuun asti vaan päättyi timeoutiin, sillä lemmikin tiedoissa oleva sukupuolen valinta ilmoitti virheestä, vaikka radio button olikin valittuna. Pienen tarkastelun jälkeen päädyttiin tulokseen, että virhe johtui yksinkertaisesti Javascriptin hitaudesta. Kuvassa 14 on esitetty virhetilanne.



Kuva 14: Pysähtynyt testiajo

Virheen välttämiseksi käytettiin Set Selenium Speed -avainsanaa, jolla jokaisen komennon suorittamisen välissä odotetaan tietty aika. Pienen kokeilun jälkeen sopivaksi viiveeksi osoittautui 0,05 sekuntia, jolloin testi ei enää keskeytynyt, mutta ei kuitenkaan merkittävästi hidastunut. Ensimmäinen testi löytyy liitteestä A.

4.3.2 Toinen versio

Toisessa vaiheessa aloitettiin testitapauksen parametrisointi. Kummallekin eläintyypille luotiin oma testitiedosto ja tehtiin erillinen Resources-kansio, joka sisältää yhteiset avainsanat common.robot-tiedostossa. Tämän lisäksi eläimen tiedot parametrisoitiin ja ne löytyvät nyt testidatasta. Parametrejä voidaan syöttää avainsanalle ja ne saadaan käyttöön avainsanan toteutuksessa [arguments]-toiminnolla. Kuten kuvan 15 esimerkistä näkyy, niin testidatan ulkoasu muuttui entistä selkeämmäksi.

```
# encoding=utf-8

*** settings ***
Library             Selenium2Library
Resource            ${CURDIR}${/}Resources${/}Keywords${/}common.robot

*** variables ***

${BREED}            Bengali
${BIRTHDAY}         12122012
${PURPOSE}          Harrastuskäyttö
${VALUE}            1500

*** keywords ***

*** test cases ***
Kissa
    Avaa Selain Ja Kirjaudu
    Valitse Kissa Ja Tayta Tiedot   ${BIRTHDAY}     ${BREED}     ${PURPOSE}      ${VALUE}
    Tayta Vakuutuksen Tiedot
    Ostoskori
    Täytä yhteystiedot
    Yhteenveto
```

Kuva 15: Kissan testitapauksen testidata

Muokkausten jälkeen testien keskimääräinen ajoaika kasvoi parilla sekunnilla, tämäkin johtui tosin ainoastaan siitä, että kummankin testin loppuun lisättiin selaimen sulkeva Close Browser -avainsana. Nyt testejä on helppo kutsua yksinkertaisesti joko

`robot kissa.robot` tai `robot koira.robot` -komennolla.

Tässä vaiheessa kokeiltiin myös muuttujien syöttöä testiä kutsuttaessa, esimerkiksi

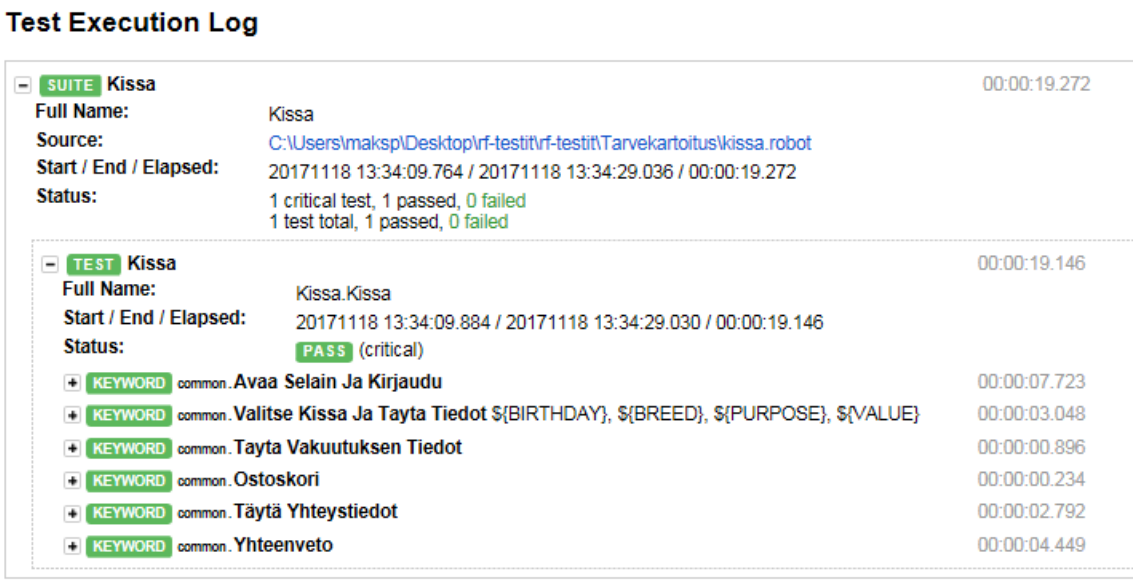
`robot --variable Breed:Maatiaiskissa kissa.robot`, jolloin kissan roduksi tulee ”Maatiaiskissa”. Variable-toiminnon käyttäminen komentoriviltä parametrinä syrjäyttää kaikki samalle muuttujalle määritetyt arvot. Useiden muuttujien alustaminen komentoriviltä on kuitenkin työlästä ja virhealtista.

4.4 Johtopäätöksiä ja jatkokehitysideoita

Lopputuloksena syntyi kummallekin eläintyypille helposti käytettävä automaattitesti. Testien lokia analysoidessa huomattiin, että suurin osa ajasta kuluu selaimen avaamiseen, ja sulkemiseen (noin. 40 % kokonaisajasta). Itse käyttöliittymän painelussa ja tietojen syötössä Robot Framework oli erittäin nopea. Tämän hetken testit käyttävät ainoastaan Chrome-selainta, mutta esimerkiksi FireFoxin ja Internet Explorerin lisäys testattaviin selaimiin ei olisi kovinkaan työlästä.

Testien kirjoittaminen oli yllättävän helppoa ja kokonaisuudessaan vei melko vähän aikaa. Jatkoa ajatellen ID:eiden käyttö elementtien tunnistuksessa sallii jopa suuriakin muutoksia käyttöliittymään, kunhan ID:t pysyvät samana, joten tehtyjen testien ylläpito vaatii todennäköisesti melko vähän työtä.

Kuvassa 16 on esitetty kissan testitapauksen esimerkkiajo. Esimerkkiajoa toistettiin 5 kertaa, jotta saatiin suoritusaikojen keskiarvo selville. Keskiarvoksi saatiin 20,35 sekuntia.



Kuva 16: Kissan testitapauksen testiloki

Automaattitestien vertailuksi suoritettiin viiden manuaalisen vastaavan käyttötapauksen suoritus webkäyttöliittymän kautta. Testien tekijä oli hyvin perillä käyttöliittymästä ja sen toiminnasta ja viiden manuaalisen testin suoritusajan keskiarvoksi saatiin 40,91 sekuntia. Automaattitestin ollessa noin kaksi kertaa nopeampi, se on myös hyvin paljon vähemmän virhealtis, sillä käsin käyttöliittymää käyttäessä virheen mahdollisuus on aina olemassa.

5. YHTEENVETO

Kandidaatintyön tarkoituksena oli automatisoida suomalaisen vahinkovakuutusyhtiön web-sovelluksen käyttöliittymätestausta Robot Framework -testiautomaatiotyökalun avulla. Luvussa kaksi käytiin yleisesti läpi ohjelmistotestauksen teoriaa eri testaustapojen kautta ja lopulta yhdistettiin neljä eri testauksen tasoa ohjelmistoprojektin eri vaiheisiin. Lisäksi esiteltiin testausautomaation käsite ja erilaisia malleja testausautomaatiotyökaluista.

Seuraavassa luvussa esiteltiin työssä käytettävä avainsanapohjainen Robot Framework, käytiin läpi sen toimintaperiaatteet ja esiteltiin hyvin keskeisenä osana oleva Robot Frameworkin käyttöliittymätestaukseen tarkoitettu Selenium 2 -kirjasto. Luvussa neljä sovellettiin teoriaa käytännössä ja kirjoitettiin valituille testitapauksille automatisoidut testit.

Testien kirjoittamiseen kului yllättävän vähän aikaa ja työn lopputuloksena saatiin automatisoitu testitapaus kahdelle yleisimmälle käyttötapaukselle – kissan ja koiran vakuutuksen ostolle. Testitapausten vakuutuksen hintaan vaikuttavat tekijät on parametrisoitu, joten testitapauksia voidaan käyttää myöhemmässä testauksessa maksunlaskennan tarkistuksessa eri parametreillä. Hankalimmaksi osoittautui käyttöliittymän tunnisteettomien (engl. id) elementtien paikantaminen. Tähän käytettiin väliaikaisena ratkaisuna xPath-funktiota, mutta käyttöliittymän myöhempää kehitystä varten kaikille elementeille on tarkoitus antaa tunniste. Lisäksi web-sovelluksen kehitystiimin kanssa syntyi ajatus, jossa automatisoidut käyttöliittymätestit liitettäisiin osaksi sovelluksen jatkuvaa kehitystä (engl. *continuous integration*), jolloin ne voitaisiin ajaa automaattisesti aina kun ohjelmaan tehdään muutoksia.

LÄHTEET

[1] C. Beust The Pitfalls of Test Driven Development, http://beust.com/weblog/2014/05/11/the-pitfalls-of-test-driven-development/. (Viitattu 20.10.2017)

[2] S. Bisht, Robot Framework Test Automation, 1st ed. Packt Publishing, GB, 2013, .

[3] M. Fewster, D. Graham, Software test automation: effective use of test execution tools, Addison-Wesley, Harlow, 1999, .

[4] T. Hammell, R. Gold, T. Snyder, Test-driven development: a J2EE example, Apress, Berkeley, CA, 2005, .

[5] W.C. Hetzel, The complete guide to software testing, second edition, 2nd ed. John Wiley & Sons, New York, 1988, .

[6] M. Katara, M. Vuori & A. Jääskeläinen, Ohjelmistojen Testaus -luentokalvot, http://www.cs.tut.fi/~tie21201/s2016/luennot/TIE-21201_2016.pdf. (Viitattu 16.10.2017)

[7] K. Kaur, S.K. Khatri, R. Datta, Analysis of various testing techniques, International Journal of System Assurance Engineering and Management, Vol. 5, Iss. 3, 2014, pp. 276-290.

[8] G.J. Myers, C. Sandler, T. Badgett, T.M. Thomas, The art of software testing, 2nd; 2 ed. Wiley, Hoboken, NJ, 2004, .

[9] R. Pressman, B. Maxim, Software Engineering: A Practitioner's Approach 8th Edition, 8th ed. 2015, .

[10] Robot Framework Foundation Robot Framework, http://robotframework.org/. (Viitattu 8.11.2017)

[11] Robot Framework Foundation Robot Framework User Guide, http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html. (Viitattu 15.10.2017)

[12] Robot Framework Foundation Selenium2Library, http://robotframework.org/Selenium2Library/Selenium2Library.html. (Viitattu 18.11.2017)

[13] E. Steegmans, P. Bekaert, F. Devos, J. Boydens, Black and White Testing: Bridging Black Box Testing and White Box Testing. 2012, https://lirias.kuleuven.be/bitstream/123456789/134277/1/Paper.pdf. (Viitattu 4.10.2017)

LIITE A: ENSIMMÄISEN VERSION TESTI

```
# encoding=utf-8

*** settings ***
Library             Selenium2Library
Library             ${CURDIR}${/}..${/}Lib${/}generate_social_security_number.py

*** variables ***
${USERNAME}      ********
${PASSWORD}      ********
${MYYMALA}       ********

*** keywords ***

*** test cases ***
Avaa Selain Ja Kirjaudu

    open browser                    http://*********/#etusivu  chrome
    Set Selenium Speed              0.05
    Set Window Size                 ${1920}     ${1080}
    wait until page contains        Tervetuloa!
    click element                   id=login_seller
    wait until page contains        Kirjaudu
    input text                      id=username         ${USERNAME}
    input text                      id=password         ${PASSWORD}
    input text                      id=pointofsales     ${MYYMALA}
    click element                   xpath=//*[@class='button float_right' and text()='Kirjaudu']


Valitse Koira Ja Tayta Tiedot
    click element                   id=elain.koira
    wait until page contains        Lemmikin tiedot
    input text                      id=animalName       RF-koira
    press key                       id=animalBirthday   12122012
    press key                       name=animalBreed      Jackrussel
    press key                       name=animalPurpose    Harrastuskäyttö
    input text                      id=animalValue      1500
    click button                    id=animalSexMale
    click button                    id=animalRegisteredNo
    click button                    id=animalHealthDeclaration1No
    click element                   xpath=//*[@class='button float_right' and text()='Seuraava']

Tayta Vakuutuksen Tiedot
    click button                    id=animalInsuredNo
    click element                   xpath=//*[@class='button float_right' and text()='Siirrä ostoskoriin']

Ostoskori
    click element                   xpath=//*[@class='button float_right' and text()='Siirry ostamaan']

Täytä Yhteystiedot
    ${SSN}=                         social_security_num
    Log                             ${SSN}
    input text                      id=ssn              ${SSN}
    click element                   xpath=//*[@class='button button--get-info float_right']
    wait until page does not contain        Asiakkaan tiedot hakematta      timeout=10
    input text                      id=phoneNumber      123321
    input text                      id=email            RF-testi@testi
    click element                   xpath=//*[@class='button float_right' and text()='Yhteenveto']

Yhteenveto
    click button                    id=legal_product
    click button                    id=legal_pi
    click element                   xpath=//*[@class='button float_right' and text()='Osta']
    wait until page contains        Kiitos
```